Searching for weak keys

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by

Mukul Purohit

to the

Department of Computer Science & Engineering

Indian Institute of Technology, Kanpur

March, 2002

14/03/02
[illegible]

Certificate

This is to certify that the work contained in the thesis entitled "***Searching for weak keys***", by *Mukul Purohit*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]

March, 2002

(Dr. Manindra Agarwal)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Acknowledgements

I take this immense opportunity to express my sincere gratitude toward my .pervisor Dr. Manindra Agrawal for his invaluable guidance. It would have never een possible for me to take this project to completion without his innovative ideas ıd his relentless support and encouragement. I consider myself extremely fortunate have had a chance to work under his supervision. Inspite of his hectic schedule he as always approachable and took his time off to attend to my problems and give .e appropriate advice. It has been a very enlightening and enjoyable experience to ɔrk under him.

I also wish to thank whole heartily all the faculty members of the Department Computer Science and Engineering for the invaluable knowledge they have imırted to me and for teaching the principles in most exciting and enjoyable way. I so extend my thanks to the technical staff of the department for maintaining an cellent working facility.

My stay at IITK was unforgettable to say the least, and the biggest reason for being my classmates of the great mtech2000 batch. A small and cute batch with rieties, lots of fun, lots of birthday treats, job parties and photo sessions in the mpus. I thank my classmates just for being such great buddies.

Abstract

ılphanumeric passwords, also called *keys*, are being used for many security related pplications. A random key could be quite secure but is difficult to remember. A ypical solution that is followed is to use an easy-to-remember seed to generate a ındom key using a Pseudo Random Generator. Keys generated using simple seeds re termed as *weak keys* for this work. In this work a key breaker for weak keys has een developed, which generates keys using Pseudo Random Generators provided ith simple seeds. The key breaker can be used for any cryptographic algorithm as ey generation process is independent of the algorithm type.

Abstract

Alphanumeric passwords, also called *keys*, are being used for many security related applications. A random key could be quite secure but is difficult to remember. A typical solution that is followed is to use an easy-to-remember seed to generate a random key using a Pseudo Random Generator. Keys generated using simple seeds are termed as *weak keys* for this work. In this work a key breaker for weak keys has been developed, which generates keys using Pseudo Random Generators provided with simple seeds. The key breaker can be used for any cryptographic algorithm as key generation process is independent of the algorithm type.

Contents

List of Figures

Chapter 1

Introduction

People have been using passwords for many applications where security is a concern. A password must be difficult to guess for anyone else but easy to remember for the user himself. People tend to choose some important date, name of family member, phone number etc. But these are not secure. In the domain of computer system security, a password is known as *key*. The approach used for passwords is also applied for the keys. But this is not secure because even if cryptographic algorithm is secure, an intruder can guess the key and disturb the security system. On the other extreme, one can use random keys, which are difficult to remember for the user as well. One solution to the problem of remembering such random keys is to store them in a file on a machine. But the machine can easily be hacked and disk can be raw-read to obtain the key file, and hence key can be obtained. For the practical purposes, a middle path is followed. Easy-to-remember string is supplied to a Pseudo Random Generator as a seed, to generate a key. However even this approach is not secure, and we will call such keys as weak keys.

To overcome the above problem of supplying easy to remember string as seeds to PRG to obtain insecure keys, one approach is to use smart cards. Smart cards to generate secure keys for cryptographic algorithms are available. A random number is supplied to a smart card, which generates a key and stores in the memory of the card. This process is executed once. Now the key is not readable from the card, even raw read can not be done. Whenever some text is to be encrypted or decrypted, it

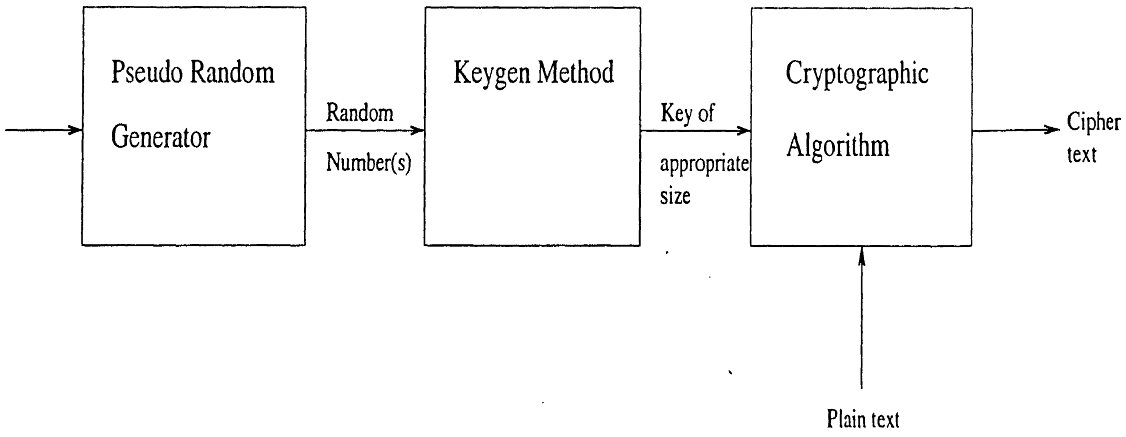


Figure 1.1: Actual Key generation process

supplied to the card as an input and corresponding decrypted or encrypted text output from the card. So the only possible security attack can be thought of is ysical theft of the card by an intruder.

In this work, a key-breaker has been implemented which takes an advantage of e above mentioned weakness. The application developed generates a key and tries decrypt the encrypted file. The process of generation of key is quite similar to e actual key generation technique as explained further. Details about the work is plained in the next section. Section three contains experimental results and it is llowed by the conclusions.

.1 Key Generation Process

rocess to generate the key for cryptographic purpose is described in the figure 1.1.

As shown in the figure, seed can be any number, and this particularly bears me relation to user, or in other words, it must be easy to remember for the user. his seed is then inputed into a Pseudo Random Number Generator, for example *ind* library function available with linux/gcc environment. Now the PRG will oduce one or more random numbers. These number(s) then undergo some simple uristics (call it keygen method) like XOR-ing of bits, rearrangement of bits etc., to produce r-bits, where r is the key length for a given algorithm. The above process can be made more complex by using a random seed instead of simple one, but then it will be difficult for the user to remember the seed.

1.2 Weak Keys

In the key generation process, the seed is an easy-to-remember string. Our approach takes advantage of this weakness and henceforth we define *weak keys* as the keys generated using simple seeds.

Chapter 2

Methodology

2.1 Our Key Generation Process

The key generation process for this work is based on brute force key generation, as seen in figure 2.1. In this approach, all possible simple seeds are covered. A simple seed is just a small number for the current application. For each seed, a key is generated using some *keygen* method, and then checked for its validity by decrypting the cipher text with the assumption that plain text is a normal english text. Since one may not be sure of the PRG, *keygen* method, or encryption algorithm, the application provides options to specify the number of possible combination for each of the parameter.

So the success of this method will broadly depend upon following issues :

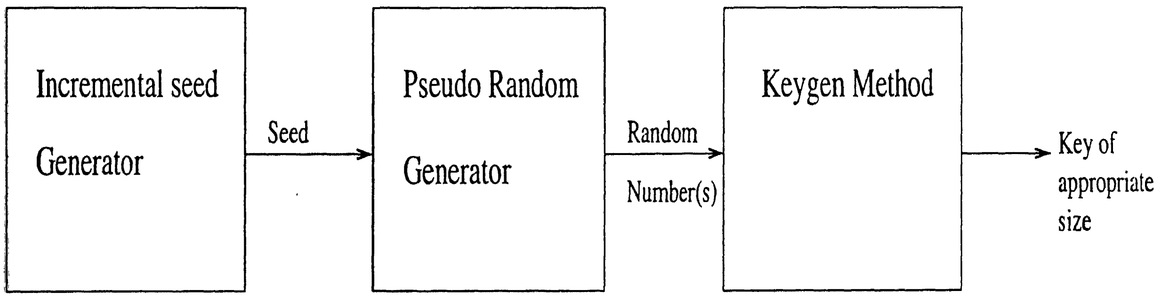


Figure 2.1: Key Generation process used

- Seed used
- Keygen method used
- Computing power available (this is to get faster results).

2.2 Design issues

In the actual implementation, besides keygen process, an interface with various cryptographic algorithms is required, so that encrypted text may be decrypted and checked if plain text is produced. So broadly there are four modules in the implementation -> seed generator, pseudo random generator, keygen method, and, cryptographic algorithm. Now we discuss the design issues for the different modules in the process.

- As described earlier, there are multiple components for each module. So a tree like structure is generated, where each leaf node represents a *computation* tuple, which can be processed individually. This tuple may have information like algorithm type, keygen type, PRG type etc. So a structure to represent such a tuple, is required. (Refer figure 2.2)
- There are various parameters required from the user. Some of them are, number of algorithms supported, number of algorithms should be tried upon, start seed, final seed etc. There are quite a few ways to obtain such information from user. One is to use global constants in a header file. But for each change, re-compilation of code is required. Also one can use arguments to the program execution, but the usage must be simple. One of the easiest is to have an interactive system. A quite efficient mechanism is to maintain some configuration files. A combination of some of the above methods can also be used.
- Since the different tuples can be executed independently of each other, they can be executed in many possible ways. If a single machine is available then some of the options are - fork a new process for each tuple, use thread library

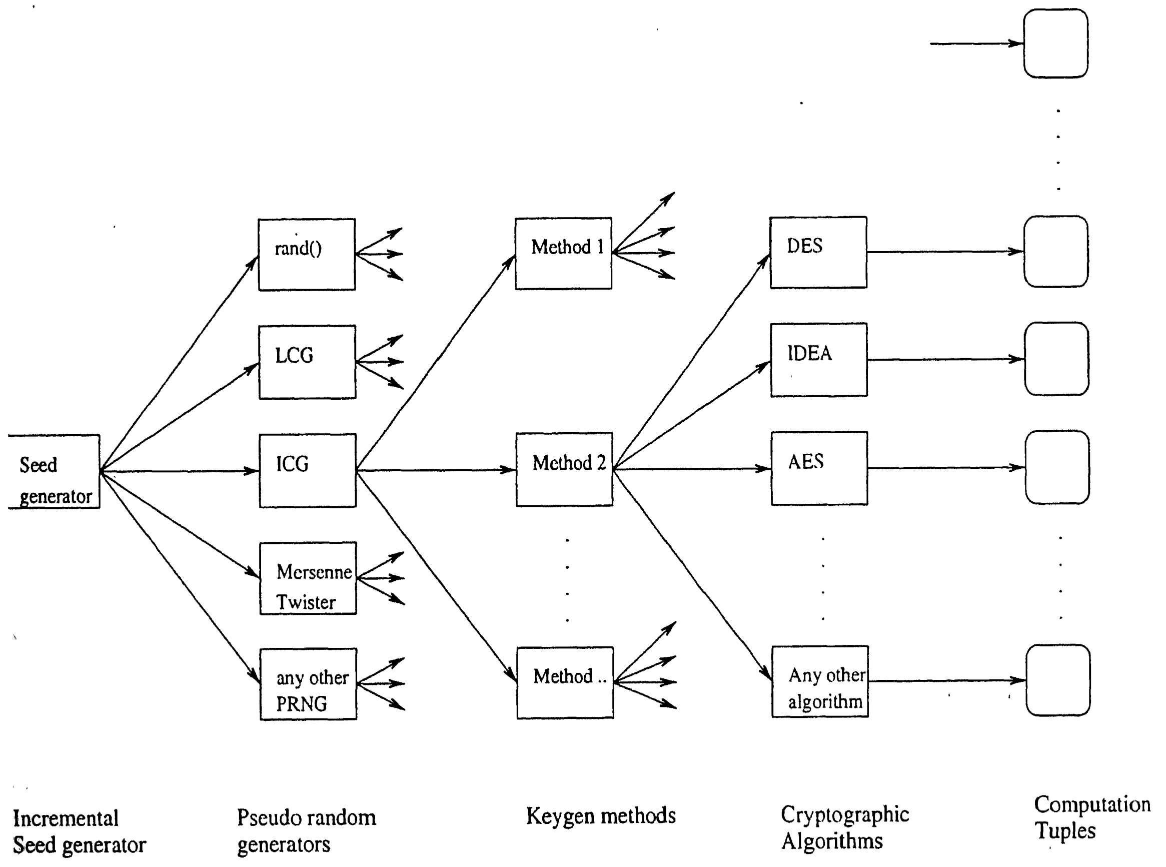


Figure 2.2: Computation tuples

instead of forking, just execute each of them serially, or use a combination of them.

- In the multiprocessor environment, one can obtain faster result, but there are some additional issues involved. For the distribution of computation, a client-server model is most appropriate, with a single client multiple server configuration. Also if a remote server fails while executing a certain tuple, due to any reason, that tuple must be reexecuted on some other machine.

- Modularity and Extensibility are two important design goals of this implementation. Extensibility is necessary as there can be new and better PRGs, new cryptographic algorithms altogether, new keygen methods could be thought of, etc. So to incorporate any new component at any of the module, there must be minimal change in the code, and also that part of code which is to be changed must be easily locate-able, a direct consequence of modular code.

- Output is also a necessary issue. Not only the success must be output, but failure output is also required. Proper output for failure cases makes the way for further execution of code with appropriate parameters.

2.3 Implementation

Now we discuss the methods actually implemented in the key breaker.

- For the purpose of input, a combination of some of the methods discussed above is used. There are some parameters like total number of algorithms supported, last seed to be tried upon, etc., which are subject to change not so frequently, so these must be in a configuration file. (refer appendix A). A sample configuration file is shown in appendix. Also information like addresses of remote machines available is kept in another configuration file. There are some parameters like number of machines available for current execution, file containing encrypted data, etc., which are subject to change on each execution. So filename is provided as argument and remaining information is input interactively.

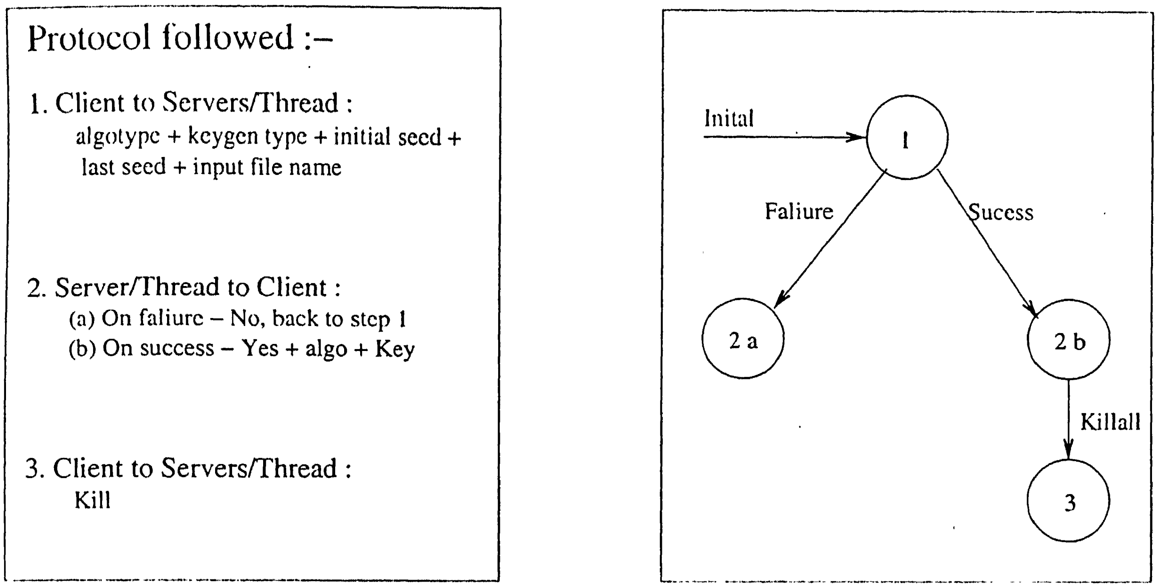
- If a single machine is available then forking new process or even running a new thread will generally not provide true optimality as this is OS dependent issue. So on a single machine, all tuples are executed serially.

- When there are multiple machines available, work must be distributed in an optimal way. Now the candidates for distribution of work are :

  - Algorithm type, (DES, AES, IDEA etc.)
  - Keygen method, (various heuristics possible)
  - PRG type, (rand, LCG, ICG, MT etc.)
  - Seed space.

In the actual implementation, the distribution is done upon first two parameters only. This can easily be extended for the other two parameters too.

As discussed before, a simple client-server model, with single client, multiple servers seems quite appropriate for this work. There is only one client on a local machine, and multiple servers on remaining $(k-1)$ machines. The server process is initiated before the client process, as in any client-server application. It will wait for the client process to make some requests. The client process, first read some data from configuration files and also reads some data from the user, and it decides that there are $(n * m)$ tuples for which computation has to be performed on k machines, where n is the number of algorithms, and m is the number of keygen methods. A tuple, consisting of <*algorithm type, keygen method type, initial seed, final seed, filename*>, is then sent to each remote machine, besides initiating a thread to perform it on the local-machine. The main thread then will wait until any of the server or computing thread sends back a message, which can either be success or failure. As soon as one of the server or the local thread reports success to the main thread of client process, this will send a KILL to all other servers and to the local thread performing the computation. On the other hand, if the server or thread exhausts all its seeds for all the PRGs listed, without any success, it will be sent the next tuple if available. If there is no tuple left for execution, then


Protocol followed :–

1. Client to Servers/Thread :
   algotype + keygen type + initial seed + last seed + input file name

2. Server/Thread to Client :
   (a) On faliure – No, back to step 1
   (b) On success – Yes + algo + Key

3. Client to Servers/Thread :
   Kill

Figure 2.3: Protocol between client and servers

the main thread will just send a kill signal to reporting server or local-thread. One by one all the servers and local thread will be killed in the order of there completion of current execution. If all the tuples are exhausted, failure is accepted gracefully. Also the mechanism to re-execute a tuple, if the currently executing machine goes down, is deployed. This is done by maintaining a flag for each tuple, if it has been executed completely and properly. In any case failure of local-machine will require restarting the whole process, but the log-file may provide with some hints about, with what parameters the execution should be initiated again. In this way the whole process works. The protocol followed between client and servers is explained in the figure 2.3.

- An absolute generic extensible implementation is quite tedious task, besides being sub-optimal. Some information about the module which must be extensible, eases the task besides achieving better optimizations. The key modules which may require extensibility are algorithm, and keygen method, which are discussed one by one.

After analyzing code for various cryptographic algorithms, it seems that these are generally implemented to provide two sets of functions. Let the first set be refereed *keyfunctions*, generically. This set will contain one or more of the functions of type *set_ the_ key(key, ...)*, which are required to set the key or make the key functional for the algorithm. The other set may be refereed as *cryptfunctions*. This set will contain one or more of the functions of type *block_ decrypt(blockin, blockout, ...)*, which are required to decrypt a block of input data into a block of output data. After this analysis, it is quite easy to provide a semi-generic interface to add a new algorithm, such that original works with the new code , without any hassles. So there are two functions provided in the code with the names given to generic functions viz *keyfunctions*, and, *cryptfunctions*. The new algorithm is added quite easily by just including a case for the new algorithm, and making the function call with the actual names of the functions provided by this algorithm, instead of these generic names. Some information about the new algorithm is also required in the configuration file, like key size, block size, etc., so that the remaining modules of the code can work correctly with the new algorithm. Finally the complete code must be recompiled and code for this algorithm must be linked to original code.

Since keygen methods are not that much complex, including new components for this module is bit easier. It can be observed that input to this module is some random number(s), and output is correct sized key. If this module is further subdivided, there may be a function for generating a bit using some heuristics which are arranged serially in appropriate data structure to give correct sized key. So a change must be made in those heuristics only. This can be performed by devising new method for generating a bit, given a random number. So in the actual implementation, a new function can be added with the name *generate_ a_ bit_ x(random_ number)* which will output either a 0 or 1. Again re-compilation of changed code is required, along with a mention in the configuration file, the current number of total-methods supported.

- As discussed above there must be provision for output not only if success is achieved but also if failure results. On success, simply the *key* is required. A log file is maintained to log various system failures, like a remote machine goes down etc., but this will also log which data set has been executed. The information contained in such a data set is more or less the tuple structure. These can be refereed in future, while execution of the whole process with new input parameters. Log file is maintained on each remote machine also, if detailed analysis is required. Since log file is maintained, a copy of key is output to this file too, so that in case of system failures an extra copy may help.

2.4 Code Re-usability

Since there were many functionalities, for which code is already available, it has been tried, to use the available code itself, instead of going for reinventing the wheel. Appendix B describes some of the functionalities of the code reused in the actual implementation.

Chapter 3

Experimental Results

All the tests have been performed on Intel Pentium-3 machines of different speeds, in the linux environment.

3.1 For known plain text

The code has been tested on five machines for input encrypted using a random key with a large seed (say 10^8) and then process is started with initial seed as 1. So time taken to go through one seed is obtained.

S No	Algorithm	Method	No of seeds	Avg Time (in seconds)	Avg Time/seed (in
1	DES	1	10^8	152671.388	1.527
2	DES	2	10^8	150132.856	1.501
3	IDEA	1	10^8	22283.045	0.222
4	IDEA	2	10^8	12982.641	0.129
5	AES(128,128)	1	10^7	29492.627	2.949
6	AES(128,128)	2	10^7	28590.056	2.859

AES(128,128) implies AES with block size 128 bit and key size 128 bit.
The keygen methods mentioned here uses some heuristics to generate appropriate

length key. First method simply generates a random number, XORs all the bits to get one bit, and repeats the process for key size times. Second method generates keysize/sizeof(randomno) random numbers and place them side by side to obtain the required key.

3.2 For unknown plain text

Range of seeds used to encrypt file	: $> 10^7$
Machines used	: 5
Number of PRGs	: 4
Number of keygen methods	: 5
Number of Algorithms	: 3

Encrypted text S No	Time (in seconds)	Algorithm
1	62672.763	DES
2	61972.045	DES
3	71759.648	IDEA
4	67495.361	IDEA
5	61458.002	DES
6	68464.569	IDEA
7	62789.884	DES
8	71147.056	IDEA
9	62178.719	DES
10	72748.007	IDEA

Chapter 4

Conclusions and Future Work

In this thesis a key breaker is developed, which thrives on a particular weakness in generation of keys, that is the tendency to use simple seeds for PRG, to generate a key. The key breaker developed applies a brute-force on the simple seeds. Simple seeds for this work are small integers. This can be further extended whence the simple seed may cover alphanumeric strings.

The key breaking is a computation intensive work, so the execution speeds play an important role for the feasibility of such an application. The key breaker is a software application and the execution speed is system dependent. With the mentioned system configuration, to execute for one seed, DES takes on an average 1.5 ms, IDEA takes 0.2 ms, and AES takes 2.8 ms, where AES is for 128 bit block size and 128 bit key length.These execution times depend upon the implementation of respective algorithms, and the code for all the algorithms is reused. Among the implementations used with this key breaker, AES seems to be an inefficient implementation.

The key breaker is quite extensible and new versions, or even new algorithms, may be plugged in easily with the remaining code.

Appendix A

Appendix – Sample Conf file

```
# This is comment line.
# All of the next five fields required. "firstseed" is default 1 so may be left #
totalmethods: 2
totalalgos: 2
firstseed: 10000000
maxseed: 100000000
totalprng: 4

##############################
algo: DES
dataTypeOfKey: "unsigned char"
sizeOfKeyType: 8
keyArraySize: 8
##sizeof block means bytes converted in one call of crypt function
dataBlockSize: 64
end

##############################
algo: IDEA
dataTypeOfKey: "unsigned short"
```

```
sizeOfKeyType: 16
keyArraySize: 8
##sizeof block means bytes converted in one call of crypt function
dataBlockSize: 64
end
```

Appendix B

Appendix – Code Reused

B.1 DES

Source and author is unknown. It is a short and precise code which provides following functions :

- *void set_the_key(int crypt, unsigned char *key, unsigned int rounds);*
  To set the key.
- *void des(unsigned char *inblock, unsigned char *outblock, int rounds);*
  To encrypt/decrypt one block of data.

B.2 IDEA

This has been authored by Richard De Moliner, of Swiss Federal Institute of Technology, Zurich. This is quite fast code and provides following functions to work upon :

- *void Idea_InvertKey (Idea_Key key, Idea_Key invKey);*
  To Invert a decryption/encryption key to encryption/decryption key.
- *void Idea_ExpandUserKey (Idea_UserKey userKey, Idea_Key key);*
  To expand a user key of 128 bits to a full encryption key.

- *void Idea_Crypt (Idea_Data dataIn, Idea_Data dataOut, Idea_Key key); Depending upon the value of 'key', it either encrypts or decrypts 'dataIn' into 'dataOut'.*

All the derived data types are defined in the code.

B.3 AES

This has been authored by Paulo Barreto and Vincent Rijmen, and was submitted to NIST during AES development effort by Joan Demen and Vincent Rijmen, the developers of Rijndael algorithm. The code has been slightly modified to make it more generic. The functions provided are :

- *int makeKey(keyInstance *key, BYTE direction, int blockLen, int keyLen, unsigned char *binkey);*
  To initializes a key.

- *int cipherInit(cipherInstance *cipher, int blockLen, BYTE mode, char *IV);*
  To initialize cipher.

- *int blockDecrypt(cipherInstance *cipher, keyInstance *key, BYTE *input, int inputLen, BYTE *outBuffer);*
  To decrypt one block of data.

All the structures are defined in the code. This code is somewhat slow, and it is hoped that in future, faster and more efficient versions will be available.

B.4 PRG

This has been authored by Otmal Lendl, and currently maintained by Josef Leylold. This is basically a collection of algorithms, for generating pseudo random numbers, implemented as a library of C functions, released under GPL. The major functions are :

- *struct prng prng_new (char *STR);*
  To create a new generator object.

- *prng_num prng_get_next_int (struct prng *G);*
  To obtain a random integer.

- *void prng_free (struct prng *G);*
  To destroy generator object.

- *void prng_seed (struct prng *G, prng_num NEXT);*
  To re-seed the generator.

Bibliography

[1] William Stallings. *Cryptography and Network Security - Principles and Practices, Second edition, Prentice Hall, 1998.*

[2] *Bruce Schneier. Applied Cryptography, Second Edition, John Wiley Sons, 1990.*

[3] *Pseudo Random Number Generator Library, statistik.wu-wien.ac.at/prng.*

[4] *IDEA Source, www.isi.ee.ethz.ch/ harpes/publications.html.*

[5] *AES Algorithm (Rijndael) Information, csrc.nist.gov/encryption/aes/rijndael.*